Module 9:

**Design pattern :** Design pattern is best practise or solution for repeating problem statement. Design pattern are proven, reusable solution to common problem that developer face while developing the application.

Why When where etc.

**SOLID Principles**

S -🡪 Single responsible principle

O-🡪 Open and Close principle

L🡪 Liskov substitution Principle

I🡪 Interface segregation Principle

D -🡪 Dependency Inversion Principle

Single -🡪 “**A class should have one and lonely one reason to change, meaning that a class should have only one job or only one purpose.**”

class Organization {

id,name,salary,desg,phone,city,state,pincode,taxAmount, panCard, accno,typeofaccount,pid,pname, typeofproject, clientid,clientName

}

Class Employee {

Id,name,salary,desg etc

Address add;

Account accno;

}

Class Address {

City,state,picode, etc }

Class Account {

Accno,typeofaccount, amount

}

Class Project {

Pid,typeproject,startdate,enddate

List<Employee> listOfEmp;

}

**Open Closed principle**

“Object or entity should be open for extension but closed for modification”

class Rectangle {

length, width, getter and setter methods

}

class Circle {

}

class AreaCalculator {

public double findArea(Rectangle tec){

logic to find the area

}

public double findCircle(Circle cc){

logic to find the area

}

}

interface Shape {

Public double calculateArea();

}

Class Circle implements Shape {}

Class Rectangle implements Shape {}

**Lishov Substitution Principle**

Creating sub class object and assign the super class reference(Run time polymorphism)

**A super class reference can hold a sub class object.**

Class Bike {

Speed() {}

}

Class Pulsar extends Bike {

Speepd() {} override

Mailage(){}

}

In main method

**Bike bb = new Pulsar();**

bb.speed() not mailage

**bb.mailage(); Error**

**Interface segregation principle**

Rathen than creating generic interface for all client application create specific interface for every client or application.

interface GeneralInterface {

public void log();

public void openDbConnection();

public void closeDbConnection();

public void openFile()

public void closeFile();

}

class LogInfo implements GeneralInterface {

Need to provide the body for all methods.

}

Class DbInfo implements GeneralInterface {

Need to provide the body for all methods.

}

Class FileInfo implements GeneralInterface {

Need to provide the body for all

}

Interface LoggerInfo {

Public void log();

}

Interface DbInfo {

Public void openConnection()

Public void closeConnection();

}

Interface FileInfo {

Public void openFile()

Public void closeFile();

}

**Dependency Inversion Principle**

“Abstraction should not depends upon implementation. Implementation should be depends upon abstraction”

interface Rbi {

withdraw();

deposit()

}

Class Hdfc implements Rbi {

}

GOF (Gang of Four): which divided into 3 parts

Creational design pattern

Deal with initialization and configuration classes and object.

1. Singleton design pattern
2. Factory method design pattern
3. Abstract factory method design pattern
4. Builder design pattern
5. Prototype design pattern.

Structural design pattern

Deal with decoupling interface and implementation of classes and interfaces.

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Behavioural design pattern

Deal with dynamic interaction among classes and objects.

1. China of responsibility
2. Command design pattern
3. Interceptor
4. Iterator (Iterator interface)
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Visitor
11. Template

Manager/Developer is a Employee

Employee has a Address (inside one class we are creating another class object is known as has relationship).

class A {}

class B {}

class Employee {

id,name,salary -🡪 primitive value

Address add = new Address();

}

class Manager extends Employee{numberOfEmp}

class Developer extends Employee{}

class ProjectManager extends Manager{}

class Address {city and state }

has a relationship

1. Association
2. Aggregation
3. Composition

Class A {

B obj1 = new B();

}

Class B {

A obj2 = new A();

}

It is a type of association but it is known a weak association ie Aggregation.

Class Employee {

Address add = new Address();

}

Class Address {

city

}

It is a type of association but it is known a strong association ie Composition.

Class Student {

StudentHistory sh = new StudentHistory();

}

Class StudentHistory {

}

UML diagram